BITS Pilani

ML SYSTEM OPTIMIZATION
Session 1

innovate achieve lead

BITS Pilani

M1: Foundations

Performance Metrics & Complexity Analysis

innovate achieve lead

BITS Pilani

1. *Time/space complexity*
   Throughput
   Latency
   Speedup (Amdahl's Law)

Time Complexity

- Time complexity measures how the **execution time of an algorithm** grows with the size of input nnn. It is independent of hardware but depends on the algorithm's structure.

  T(n)=time taken to process input of size n

- Helps **predict performance** for large inputs.
- Allows comparison between **different algorithms** solving the same problem.
- Guides **optimization** in high-performance computing.

Asymptotic Notations

Big O (O): Upper bound, worst-case time

T(n) = O(f(n)) ⟹ Algorithm will take at most c·f(n) time for large n

Omega (Ω): Lower bound, best-case time

T(n) = Ω(f(n)) ⟹ Algorithm will take at least c·f(n) time for large n

Theta (Θ): Tight bound, average-case time

T(n) = Θ(f(n)) ⟹ Algorithm grows exactly like f(n)

Time Complexity

Algorithm	Best Case	Average Case	Worst Case
Linear Search	Ω(1)	Θ(n)	O(n)
Binary Search	Ω(1)	Θ(log n)	O(log n)
Merge Sort	Ω(n log n)	Θ(n log n)	O(n log n)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)

Common Growth Rates

- **O(1):** Constant time – independent of n
- **O(log n):** Logarithmic – e.g., binary search
- **O(n):** Linear – e.g., linear search, sum of array
- **O(n log n):** Log-linear – e.g., merge sort, quicksort average case
- **O(n^2):** Quadratic – e.g., bubble sort, insertion sort
- **O(2^n), O(n!):** Exponential / factorial – impractical for large n

Time Complexity

Linear Search: Search for element in array of size n

$$T(n)=\text{comparisons made}=n \text{ (worst case)}$$
$$T(n)=O(n)$$

Binary Search: Search in sorted array

$$T(n)=1+1+\cdots+\log_2 n=O(\log n)$$

Recursive Algorithms

Time complexity can also be **recursively defined**. Example: Merge Sort

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$
$$T(n)=2\cdot T(2n)+O(n)$$

Space Complexity

Space complexity measures **how much memory an algorithm requires** relative to input size n. It includes:

1. **Input space:** Memory needed to store inputs
2. **Auxiliary space:** Temporary space used by algorithm
3. **Function/recursion stack:** Memory for recursion calls

$$S(n)=S_{input}+S_{auxiliary}+S_{stack}$$

Why Space Complexity Matters

- Helps predict **memory requirements** for large datasets.
- Prevents **memory overflows** in HPC and parallel systems.
- Guides algorithm selection for memory-constrained systems.

Work Integrated Learning Programmes

Space Complexity

Linear Search: Only a few variables used for index and element comparison.

$$S(n)=O(1)(constant\ space)$$

Merge Sort: Uses an auxiliary array for merging.

$$S(n)=O(n)(extra\ array\ of\ size\ n)$$

Quick Sort (in-place): Recursion stack used: log n depth on average

$$S(n)=O(log\ n)(average\ case)$$

Matrix Multiplication: For two n×n matrices:

$$S(n)=O(n^2)(stores\ output\ matrix)$$

innovate achieve lead
Work Integrated Learning Programmes
BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Space Complexity

If an algorithm uses:

- k scalar variables → each takes constant space → total **O(k)**
- n input elements → space grows with input size → **O(n)**
- Recursion depth d → each call uses stack space → total **O(d)**

Then:

$$S(n)=O(n + k + d)$$

All these memory usages happen together, so we **add them**. In Big-O, we keep the dominant growing term. If **k is constant** and **d ≤ n**, then:

$$S(n)=O(n)$$

because **n grows fastest and dominates the total space**.

Throughput

Throughput is the **number of tasks or operations completed per unit time**. It measures **system-wide productivity**.

$$\text{Throughput} = \frac{\text{Number of Tasks Completed}}{\text{Total Time}}$$

Example (Sequential System):

- CPU processes 1 task every 0.25 sec
- For 100 tasks:

$$\text{Throughput} = \frac{100}{100 \times 0.25} = 4 \text{ tasks/sec}$$

In Parallel Systems:

- Throughput can increase with **more cores/processors**
- Ideal case (linear scaling):

$$\textbf{\textit{Throughput}}_{\textbf{\textit{parallel}}} = N \times \textbf{\textit{Throughput}}_{\textbf{\textit{single core}}}$$

where N = number of cores

Latency

Latency is the **time taken to complete a single task from start to finish**.

Latency = Finish Time − Start Time

Example:

- Task A starts at 0 sec, finishes at 2 sec → Latency = 2 sec
- Task B starts at 1 sec, finishes at 3 sec → Latency = 2 sec

Important Note:

- Low latency ≠ high throughput (especially in parallel systems)
- Example: Batch processing can increase throughput but individual task latency may remain high

Relationship Between Throughput and Latency

Little's Law (Queueing Theory)

$$L = \lambda \cdot W$$

Where:

- L = Average number of tasks in the system (system load)
- λ = Arrival rate (tasks/sec) → closely related to **throughput**
- W = Average time spent in the system → **latency**

Rearranging:

$$W = \frac{L}{\lambda}$$

- If arrival rate λ increases
- But system resources (CPU, memory, cores) remain fixed

As tasks begin to queue, the load (L) increases, which in turn increases the latency (W).

Relationship Between Throughput and Latency

Increasing throughput does **not automatically reduce latency**.

If system utilization approaches 100%:

- Queue length grows rapidly
- Waiting time increases
- Latency increases nonlinearly

In many queue models (e.g., M/M/1):

$$W = \frac{1}{\mu - \lambda}$$

Where:

- μ = service rate
- As λ→μ

W→∞

So latency grows dramatically near saturation.

Core Performance Metrics

$$\text{Throughput} = \frac{\text{Tasks Completed}}{\text{Total Time}} \quad (\text{tasks/sec})$$

$$\text{Latency} = \text{Task Completion Time} - \text{Task Arrival Time} \quad (\text{seconds})$$

$$\text{Efficiency} = \frac{\text{Throughput}_{\text{actual}}}{\text{Throughput}_{\text{ideal}}}$$

In parallel systems:

$$\text{Efficiency} = \frac{\text{Speedup}}{p}$$

Where:

$$\text{Speedup} = \frac{T_1}{T_p}$$

- T_1 = execution time on 1 processor
- T_p = execution time on p processors
- p = number of processors

Speedup

Speedup measures how much faster a parallel system performs compared to serial execution.

$$S = \frac{T_{serial}}{T_{parallel}}$$

Where:

- T_{serial} = execution time on **one processor**
- $T_{parallel}$ = execution time on **N processors**

Example

- Single-core time: T_{serial} = 100 sec
- 4-core time: $T_{parallel}$ = 30 sec

Work Integrated Learning Programmes

Speedup

- The system is **3.33 times faster** using 4 processors.
- Ideal speedup for 4 processors would be:

$$S_{ideal} = 4$$

Since 3.33 < 4, the system has some overhead (communication, synchronization, imbalance).

Important Insight

$$S \leq N$$

- If S = N → Perfect linear speedup
- If S < N → Parallel overhead exists
- If S > N → Superlinear speedup (rare, due to cache effects)

Work Integrated Learning Programmes

Amdahl's Law

Amdahl's Law predicts the **maximum theoretical speedup** of a program when only part of it can be parallelized.

Assumptions

- P = fraction of program that can run in parallel
- 1−P = fraction that must run sequentially
- N = number of processors

Parallel Execution Time

Total serial time = T_{serial}

Sequential portion time: $(1-P)T_{serial}$

Parallel portion time (divided among N processors):

$$\frac{PT_{serial}}{N}$$

So total parallel time:

$$T_{parallel} = T_{serial}\left[(1-P) + \frac{P}{N}\right]$$

Speedup

$$S(N) = \frac{T_{serial}}{T_{parallel}} = \frac{1}{(1-P) + \frac{P}{N}}$$

Amdahl's Law

1. The **sequential part (1 − P)** must always execute on a single processor. It cannot be reduced by adding more processors.
2. The **parallel part (P)** gets divided across N processors, so its time reduces as N increases.
3. However, even if N→∞:

$$S_{max} = \frac{1}{1-P}$$

This means the **sequential fraction limits the maximum speedup**.

Example:

If:

- P=0.9 (90% parallelizable)

Then:

$S_{max}=1 / 0.1 = 10$

Even with unlimited processors, speedup cannot exceed 10.

Amdahl’s Law

Given:

- Parallel fraction: P = 0.8
- Sequential fraction:1− P = 0.2
- Number of processors: N = 4

Apply Amdahl’s Law

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

Substitute values:

$$S(4) = \frac{1}{0.2 + \frac{0.8}{4}}$$

$$S(4) = \frac{1}{0.2 + 0.2}$$

$$S(4) = \frac{1}{0.4}$$

$$S(4) = 2.5$$

Even though 4 processors are used, the speedup is only **2.5×**, not 4×

Why?

- The 20% sequential portion must still run on one processor.
- Only the 80% portion benefits from parallel execution.
- This sequential part becomes a limiting factor.

Amdahl’s Law

If total serial time = 100 sec:

- Sequential part (20%) = 20 sec
- Parallel part (80%) = 80 sec

With 4 processors:

Parallel time: 80 / 4 = 20 sec

Total parallel execution time: 20 + 20 = 40 sec

Speedup: 100 / 40 = 2.5

The sequential portion (1−P) restricts scalability. Adding more processors improves performance, but the improvement is limited by the non-parallelizable part of the program.

Work Integrated Learning Programmes

Amdahl's Law

Suppose a video rendering task takes **200 minutes** on a single processor.

- 70% of the work (frame rendering) can be parallelized

  $P = 0.7$

- 30% (final encoding + synchronization) is sequential

  $1 - P = 0.3$

- Number of processors:

  $N = 8$

Apply Amdahl's Law

$$S(8) = \frac{1}{(1-P) + \frac{P}{N}}$$

$$S(8) = \frac{1}{0.3 + \frac{0.7}{8}}$$

$$S(8) = \frac{1}{0.3 + 0.0875}$$

$$S(8) = \frac{1}{0.3875}$$

$$S(8) \approx 2.58$$

Work Integrated Learning Programmes

Amdahl’s Law

Original time = 200 minutes

Speedup ≈ 2.58

New execution time:

$$T_{parallel} = \frac{200}{2.58} \approx 77.5 \text{ minutes}$$

Why Not 8× Speedup?

Even though 8 processors are used:

- The 70% rendering part is divided across 8 cores.
- But the 30% encoding part must still run sequentially.

Work Integrated Learning Programmes

Amdahl’s Law

Sequential time:

$$0.3 \times 200 = 60 \text{ minutes}$$

That 60 minutes cannot be reduced by adding more processors.

In real systems like video rendering:

- Parallel tasks (frame processing) scale well.
- Sequential tasks (final merge, encoding) limit total speedup.

So even with powerful HPC clusters, performance is bounded by the sequential bottleneck.

This is exactly what Amdahl’s Law predicts.

Maximum Speedup

We know:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

To find the **maximum possible speedup**, take the limit as the number of processors becomes very large:

Apply the Limit $\lim_{N \to \infty} S(N)$

As:

$$N \to \infty$$

The term:

$$\frac{P}{N} \to 0$$

So the speedup formula becomes:

$$S_{max} = \frac{1}{(1-P)}$$

Maximum Speedup

Suppose total serial execution time = 100 seconds.

- Parallel part (80%) = 80 sec
- Sequential part (20%) = 20 sec

If we use infinite processors:

- Parallel part becomes ≈ 0 sec
- Sequential part remains 20 sec

So total time ≈ 20 sec.

Speedup: 100 / 20 = 5

Even with unlimited processors, the execution time **cannot go below 20 seconds**, because that portion cannot be parallelized.

Work Integrated Learning Programmes

Efficiency

Efficiency measures how well the processors are utilized during parallel execution.

$$E(N) = \frac{S(N)}{N}$$

Where:

- S(N) = speedup with N processors
- N = number of processors

It tells us what fraction of each processor's capacity is effectively used.

Using Amdahl's Law

We know:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

Substitute into efficiency:

$$E(N) = \frac{1}{N\left[(1-P) + \frac{P}{N}\right]}$$

Simplify:

$$E(N) = \frac{1}{N(1-P) + P}$$

Efficiency

Given:

- P = 0.8
- N = 4

Previously calculated speedup:

S(4) = 2.5

Efficiency:

- E(4) = 2.54 / 4
- E(4) = 0.625
- E(4) = 62.5%

Although 4 processors are used:

- The system achieves only 2.5× speedup.
- So each processor is effectively utilized at 62.5%.

Although 4 processors are used:

- The system achieves only 2.5× speedup.
- So each processor is effectively utilized at 62.5%.

- The remaining 37.5% is wasted due to the sequential part (1−P) of the code.

Why Efficiency < 1

- Sequential fraction of the program (1 − P = 0.2) cannot be parallelized.
- Adding more processors **does not speed up sequential tasks**, so efficiency drops.
- With more processors, efficiency **keeps decreasing**, illustrating **diminishing returns**.

Efficiency

Why Efficiency Decreases as N Increases

From:

$$E(N) = \frac{1}{N(1-P)+P}$$

As N incre

- The term N(1−P) increases.
- Denominator increases.
- So efficiency decreases.

Unless:

P→1

Meaning the program is almost fully parallelizable

Work Integrated Learning Programmes